Modern Engineering

Day 3

Prudential Financial

GENERAL ASSEMBLY

Whilst we wait....

In chat write:

A book/movie/TV series that will blow our mind.

SQL

LESSON ROADMAP

Filtering and Aggregating Data

JOINing Tables

Superstore Lab

Carmen Sandiego Lab

MEF MODULE 1 DAY 3: SQL and DynamoDB

Schedule	
9:00–9:15 am	Welcome and Warm-Up
9:15–11:00 am	**Filtering and Aggregating Data**
11:00 am–12:30 pm	**JOINING Tables**
12:30–1:30 pm	Lunch
1:30-2:30 pm	**JOINING Tables continued**
2:30–4:50 pm	**Carmen Sandiego SQL Lab**
4:50–5:00 pm	Bring It Home

LEARNING OBJECTIVES

1. Explore a data set using exploratory data analysis
2. Sort and segment query results to generate insights.
3. Filter query results based on multiple conditions
4. Combine data from multiple tables in complex queries.

SQL and DynamoDB

Filtering and Aggregating Data

Our Goal For This Section

- Sort and segment query results to generate insights.

Filtering and Aggregating Data

Aggregate Functions in SQL

Guided Walk-Through:

Navigating the Superstore Data Set

We'll be working with a single data set throughout the day to apply what we're learning. Let's go through the following steps to get started:

1. Connect to the SQL database that we'll be using for this unit.
   a. Click this link to access your nearest host browser.

Username: analytics_student@generalassemb.ly
Password: analyticsga

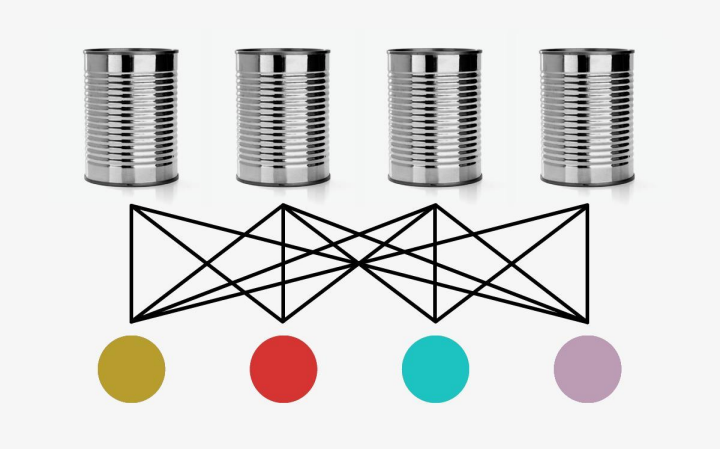
2. Explore the functions of the client software (execute, stop, save, new query).
3. Look at the first and last 100 rows of the data from the tables using the menus.
4. Review how the column properties are defined using the menus.

Aggregate Functions

In SQL, aggregate functions help summarize large quantities of data and...

- Produce a single value from a defined group.
- Operate on sets of rows and return results based on **groups of data**.

The most commonly used aggregate functions are **MIN**, **MAX**, **SUM**, **AVG**, and **COUNT**.



Guided Walk-Through:
Adding Aggregates to SELECT

Aggregate functions fit into the SELECT statement just like unaggregated columns:

```
SELECT SUM(col1)
FROM table;
```


Guided Walk-Through:

Validating Data With COUNT

COUNT is a basic aggregation function that counts the number of rows returned.

Here are two popular use cases:

- **COUNT(*)**: Counts all rows returned by the query.
- **COUNT(field)**: Counts all rows where the field is not **NULL**.

```
SELECT COUNT(*)
FROM orders
WHERE sales > 100;
```

Counts all rows of orders more than $100.

Partner Exercise:

Aggregate Functions Matchmaking

10 minutes

With your chat partner, refer to the scenarios on the left and connect them to the aggregate functions on the right that will return the results you need.

1. The number of orders placed on a specific date.
2. The highest profit margin in the consumer segment.
3. The typical quantity of copiers sold in the Central region.
4. The number of customers placing orders from Madhya Pradesh, India.
5. The lowest discount given in the furniture category.

COUNT: Counts how many values are in a particular column.

COUNT DISTINCT: Counts how many unique values are in a particular column.

SUM: Adds together all of the values in a particular column.

AVG: Calculates the average of a group of selected values.

MIN/MAX: Return the lowest and highest values in a particular column, respectively.

Partner Exercise:

Let's Try It!

20 minutes

Now that you have a good sense of what each aggregate function does, review the prompts from the previous slide and try writing out the query for each.
At a minimum, your queries should include SELECT and FROM and an aggregate function. For example:

Ready, set, go!

```
SELECT SUM(sales)
FROM orders;
```


Filtering and Aggregating Data

GROUP BY and HAVING

Clauses for Aggregate Functions

Aggregate functions are also used in these clauses:

- **GROUP BY** indicates the dimensions by which you want to group your data (e.g., a category that you want to sort into subgroups).
- **HAVING** is used to filter measures you've aggregated (e.g., to filter a SUM of more than a certain value).

Where They Live in a Query

SELECT picks the columns.

FROM points to the table.

WHERE puts filters on rows.

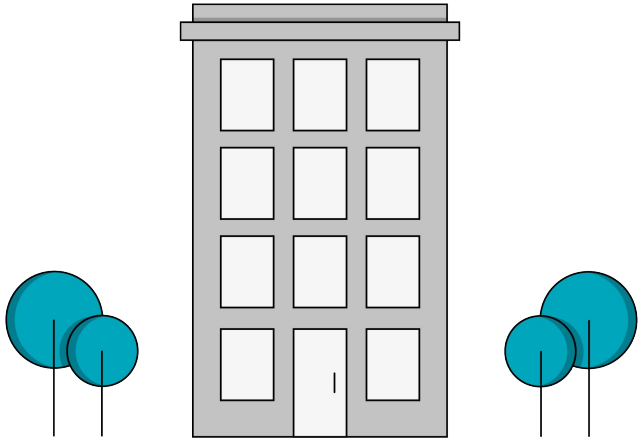
GROUP BY aggregates multiple rows, based on one or more aggregate functions (MIN, AVG, etc.).

HAVING filters aggregated values ***after*** they have been grouped.

ORDER BY sorts the results.

LIMIT limits results to the first **n** rows.

The Query Building



Partner Exercise:

Let's Build Together!

10 minutes

With your chat partner, build the following queries with GROUP BY and HAVING:

```
SELECT segment,
    COUNT(*) AS num_sales
FROM customers
GROUP BY segment
```

```
SELECT segment,
    COUNT(*) AS num_sales
FROM customers
GROUP BY segment
HAVING COUNT(*) > 300
```

- How many results do you get with the **GROUP BY** statement?
- How many results do you get with the **HAVING** statement included?

Guided Walk-Through:
Aggregating Data With GROUP BY and HAVING

Superstore wants an order discount analysis to identify average order quantity and amount by discount level. To write our query, we'll use:

1. **WHERE** to filter discount levels greater than 15%.
2. **GROUP BY** in our query to aggregate quantity and sales.
3. **HAVING** to filter discount levels above an average sales threshold.

```
SELECT discount, ROUND(AVG(quantity), 2) AS "qty", AVG(sales)::money as "sales"
FROM orders
WHERE discount > 0.15
GROUP BY discount
HAVING AVG(sales) > 500
ORDER BY 3 DESC
```


Solo Exercise:

Over to You

5 minutes

Follow the instructions in https://git.generalassemb.ly/ModernEngineering/raw-sql-superstore/blob/main/lab-3.md.

Solo Exercise:

Over to You | Solution

Here is what your end query might look like:

```
SELECT category, COUNT(*) as count_of_products
FROM products
WHERE
  product_name ILIKE '%computer%'
  OR product_name ILIKE '%color%'
GROUP BY 1
HAVING COUNT(*) > 100
ORDER BY 2 DESC
LIMIT 10;
```


Filtering and Aggregating Data in SQL

Filtering Dates

Dates are in a “YYYY-MM-DD-HH-MM-SS” format.

- **<, <=, >, >=** allow you to filter date fields on a dateline, similar to how we have been using these operators on a number line.
- **BETWEEN** will allow you to filter by segments of time.

Guided Walk-Through:
Filtering Dates

Superstore wants to know: 1) How many orders we have in 2020 and beyond? 2) How many orders did we have in the Q1 2019? To write our query, we'll use:

1. **WHERE** to filter order_date to dates to 2020 and beyond.
   WHERE order_date > '2019-12-31'

1. **WHERE** to filter the Q1 2019 date segment.
   WHERE order_date BETWEEN '2019-01-01' AND '2019-03-31'

```
SELECT *
FROM orders
WHERE order_date BETWEEN '2019-01-01' AND '2019-03-31';
```

Aggregating Dates

Aggregate functions for dates allow us to focus on parts of time (years, months or seconds.)

- **DATE_PART** used in the WHERE part of the query filters date fields by specific parts of the date (year, day, second, etc.).
- **DATE_PART** used in the SELECT part of the query in tandem with an aggregate function will create date groups (e.g., Jan 2019, Jan 2018, and Jan 2021 would be grouped as “January”).
- **DATE_TRUNC** used in the SELECT part of the query truncates the specified date to the accuracy specified by the DATE_PART. Together with an aggregate function, it creates subgroups (e.g., Jan 2019, Jan 2018, Jan 2021).

Guided Walk-Through:
Aggregating Dates

Superstore wants to know: 1) How many sales in 2019? 2) What is our all-time most profitable month? 3) Which month/year was our most profitable?

1. **DATE_PART** in WHERE will allow you to filter by date parts.
2. **DATE_PART** in SELECT will create groups.
3. **DATE_TRUNC** in SELECT will create subgroups.

```
1)  SELECT COUNT(*) FROM orders WHERE DATE_PART('year', order_date) = 2019;

2)  SELECT DATE_PART('month' ,order_date), SUM(profit) FROM orders GROUP BY 1
    ORDER BY 2 DESC;

3)  SELECT DATE_TRUNC('month', order_date), SUM(profit) FROM orders GROUP BY 1
    ORDER BY 2 DESC;
```


Solo Exercise:

Over to You | Dates

20 minutes

Follow the instructions in https://git.generalassemb.ly/ModernEngineering/raw-sql-superstore/blob/main/lab-4.md.

Solo Exercise:

Over to You | Solution

Here is what your end queries should look like:

1) **What was our most profitable month in 2019?**

```
SELECT DATE_PART('month', order_date), SUM(profit) FROM orders WHERE
DATE_PART('year', order_date) = 2019 GROUP BY 1 ORDER BY 2 DESC;
```

1) **What was our least profitable month of all time?**

```
SELECT DATE_PART('month', order_date), SUM(profit) FROM orders GROUP BY 1
ORDER BY 2 ASC;
```

1) **Which year had the highest average sales?**

```
SELECT DATE_PART('year', order_date), AVG(sales) FROM orders GROUP BY 1
ORDER BY 2 DESC;
```

Solo Exercise:

Over to You | Solution (Cont.)

Here is what your end queries should look like:

4) **Of months where we've had more than $20K in profit, which had the most transactions?**

```
SELECT DATE_TRUNC('month', order_date), COUNT(*) FROM orders GROUP BY 1
HAVING SUM(profit) > 20000 ORDER BY 2 DESC;
```


Discover SQL

JOINing Tables

The Structure of SQL

Querying Databases

Filtering and Aggregating Data

JOINing Tables

And Finally

- Combine data from multiple tables in complex queries.

Celebrating Table Togetherness

One 2019 study found that most companies with 1,000 employees or more are pulling from 400+ data sources for business intelligence. In fact, more than 20% of the organizations reported drawing from a whopping 1,000 or more data sources.

So, let's get comfortable bringing that data together!

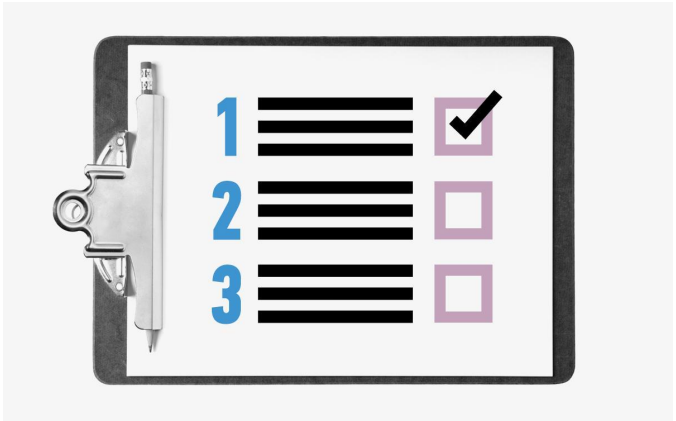
Image source: The A Team Meme Generator

Discussion:

What Could Possibly Go Wrong?

You've handled a data set or two before.
Let's make a list that addresses the following:

- What could go wrong when combining two or more data sets?
- What might you want to have control over?



JOINing Tables

Combining Data in SQL

JOINs and UNIONs

In SQL, there are two primary methods for bringing data together:

A **JOIN** combines **columns** from tables using common unique identifiers (keys).

A **UNION** combines **rows** of *similar* data.

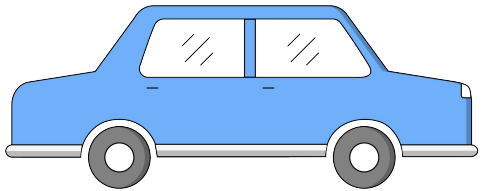
JOINs

A **JOIN** combines columns from multiple tables using a common unique identifier or “key.”

drivers		
id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	4
4	Ali	5

vehicles	
id	vehicle_name
1	Explorer
2	Civic
3	Corolla
4	Impala

id	name	vehicle_id	vehicle_name
1	Janet	3	Corolla
2	Emily	4	Impala
3	Yoko	4	Impala

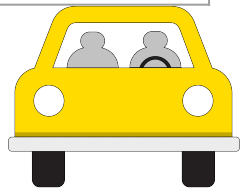


UNIONs

A **UNION** combines rows from multiple tables with similar data to create a new set. Using “UNION” removes duplicates when combining the two tables.

carpoolers		
id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	4

monthly_parkers		
id	name	vehicle_id
2	Emily	4
4	Ali	5
5	Ray	1



id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	4
4	Ali	5
5	Ray	1

Where They Live in a Query

SELECT picks the columns.

FROM points to the table.

WHERE puts filters on rows.

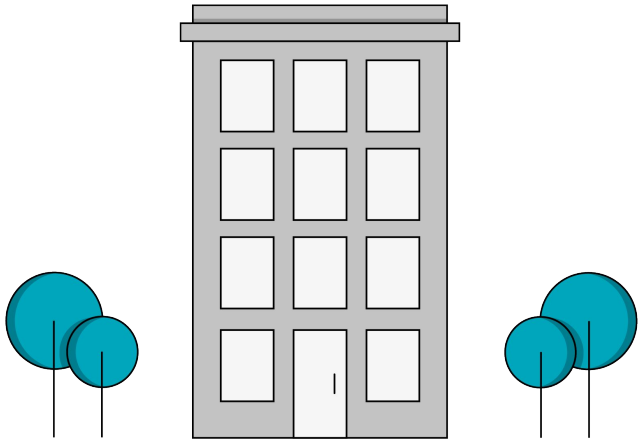
GROUP BY aggregates across values of a variable.

HAVING filters aggregated values ***after*** they have been grouped.

ORDER BY sorts the results.

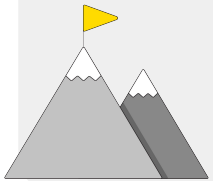
LIMIT limits results to the first **n** rows.

The Query Building



SQL Wants to Be Normal

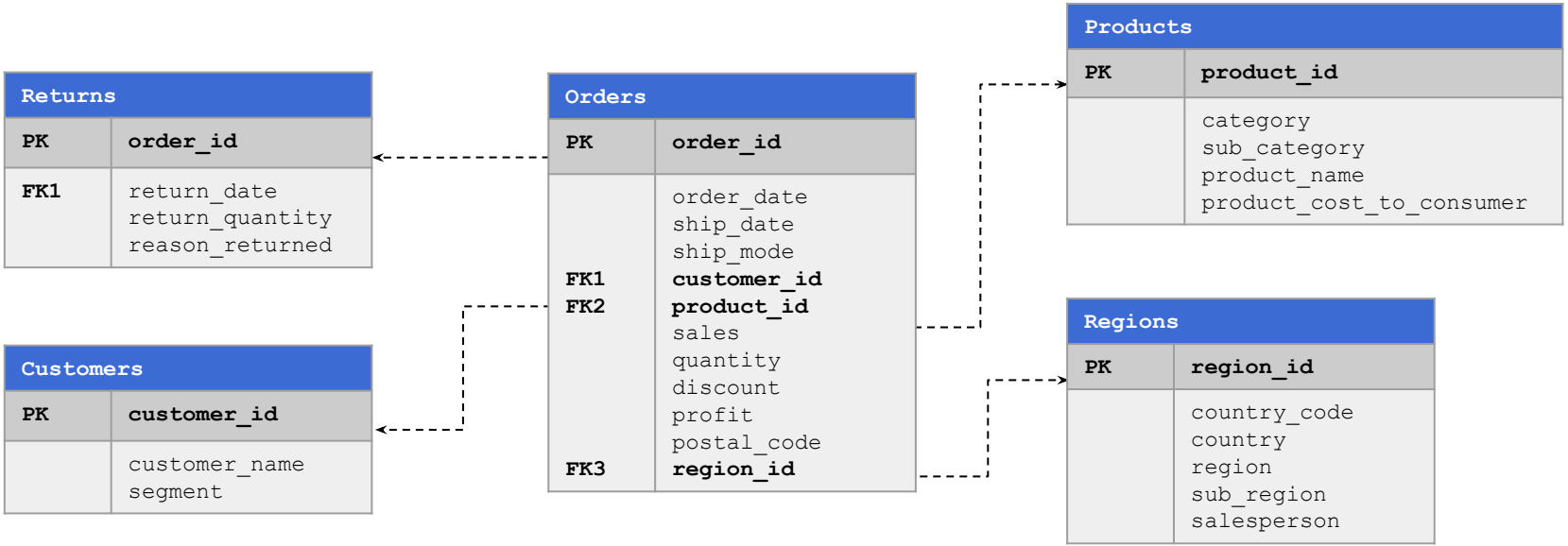
A normalized database will seek to **separate data across multiple tables** that are related to each other by keys. This reduces redundancy and memory footprint and improves speed.



SQL queries are most performant (in terms of memory and speed) when tables are **NARROW** (few columns) and **TALL** (many rows). This is where JOINs and UNIONs come into play!

Discussion:
Where Are Our Keys?

Take a look through our five tables. Which columns would we use to connect these tables?

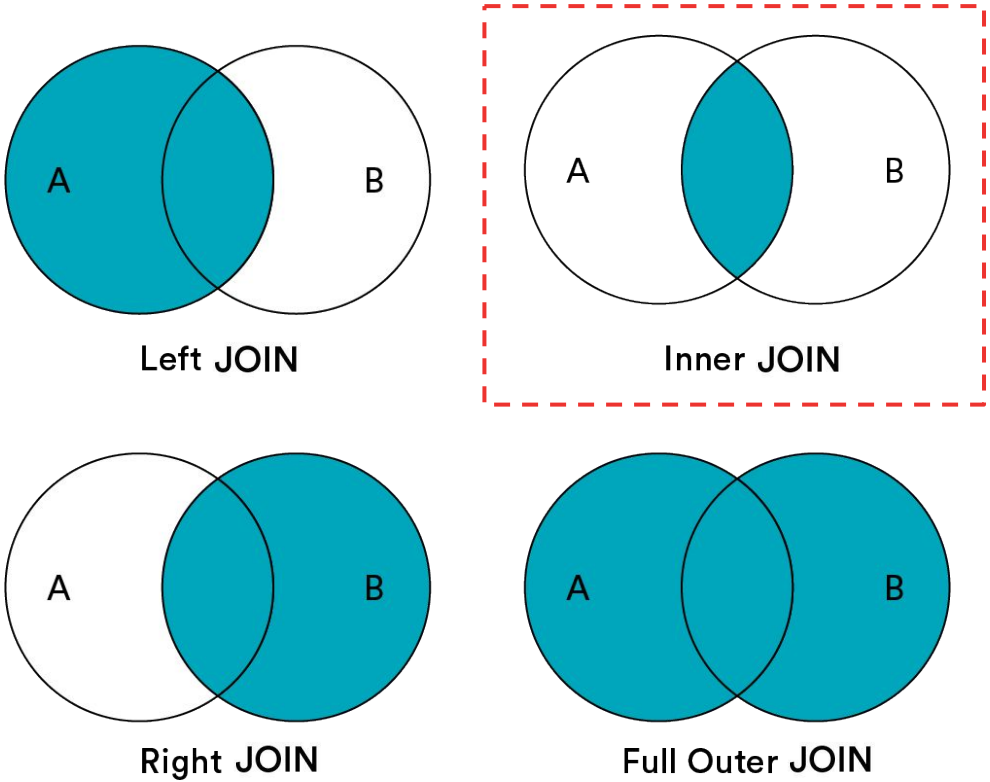


JOIN Syntax

```
SELECT
    orders.sales,
    regions.region
FROM
    orders
JOIN
    regions
ON
    orders.region_id =
    regions.region_id
```

1. Designate columns we want returned, specifying the table from which they came.
2. Name the **primary table** from which we're pulling data.
3. Name the **secondary table** from which we're pulling data.
4. Specify the **key** to JOIN these two tables.

Types of JOINs



INNER JOIN is the same thing as JOIN.

Image source: GA Instruction

Guided Walk-Through:
Let's Create a JOIN!

With the global expansion of Superstore, your sources of reliable data are also growing. That's good news, right? For the most part, yes, but...

The high volume of data can also make referencing tricky and error-prone. Just in time, you got a request from your *super* boss asking you to **identify returns by reason**. This requires you to pull and combine data from these two tables:

Orders		
order_id	**order_date**	**ship_date**
AE-2016-1308551	2016-09-28	2016-10-02
AE-2016-1522857	2016-09-04	2016-09-09

Returns		
order_id	**return_date**	**reason_returned**
AE-2019-1711936	2019-12-14	Not Given
AE-2019-2092798	2019-11-29	Not Given

Guided Walk-Through:
And So, a JOIN Is Born

```
SELECT DISTINCT
  orders.order_id
  , orders.order_date
  , returns.reason_returned
FROM orders
JOIN
returns ON orders.order_id =
returns.order_id
LIMIT 2;
```

Orders

order_id	order_date	ship_date
AE-2016-1308551	2016-09-28	2016-10-02
AE-2016-1522857	2016-09-04	2016-09-09

Returns

order_id	return_date
AE-2019-1711936	2019-12-14
AE-2019-2092798	2019-11-29

JOIN Result

order_id	order_date	return_date
AE-2016-1308551	2016-09-28	2019-12-14
AE-2016-1522857	2016-09-04	2019-11-29

Working With Long Table Names

What if you're frequently referencing tables with names like this in your query?

Sales_With_Discount_Transaction_History

Imagine adding that to a column name twice as long! The solution?

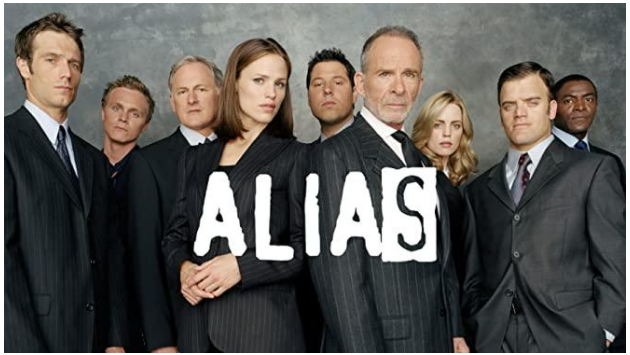


Image source: Imgflip

Shortcuts | Using an Alias

An **alias** is a shorthand name given to tables (or columns in a table) that you intend to reference repeatedly.

When creating a JOIN, each table or column can have an alias. Each column is then connected to the table by the alias.

table1 a ➔ table1 uses the alias a.

a.column4 ➔ column 4 is connected to table1 by the alias a.

Alias Syntax

Aliases are user-defined and designated in the FROM statement immediately following the table or column name.

Take a look at the syntax below. Notice that AS is in brackets because it is optional — you don't need it to designate an alias.

Alias for tables:

```
table_name [AS] alias_name
```

Alias for columns:

```
column_name [AS] alias_name
```


Guided Walk-Through:
Aliases in a Query

```
SELECT
        orders.order_id,
        orders.order_date,
        returns.return_date

FROM
        orders

INNER JOIN returns

ON orders.order_id =
returns.order_id;
```

Let's use an alias in this query from earlier. First, designate the aliases in **FROM**.

- The Orders table will be a.
- The Returns table will be b.

Next, specify the connection, by column name, on which you want to link tables:

- **ON** a.column_name = b.column_name with alias for source table.
- **USING**(column_name) only if the columns have same name in each table.

Guided Walk-Through:

Aliases in a Query | Solution

```
SELECT
        a.order_id,
        a.order_date,
        b.return_date

FROM
        orders a

INNER JOIN returns b

ON a.order_id = b.order_id;
```

This is what your query should look like with an alias for each table. Keep in mind that:

- The renaming is only temporary, and table name does not change in the original database.
- Aliases work well when there are multiple tables in a query.

Wireframing JOINs | Single Tables

You may find drawing out tables (like below) can help you conceptualize how you plan to JOIN them. Remember, wireframes do not have to be super detailed.

Primary Table **ON** **Secondary Table**

orders o	
order_id	**customer_id**
AE-2016-1308551	JR-16210
AE-2016-1522857	KM-16375
....	....

customers c	
customer_id	**customer_name**
JR-16210	Justin Ritter
KM-16375	Katherine Murray
....	....

Solo Exercise:

JOINing Single Tables

10 minutes

Follow the instructions in https://git.generalassemb.ly/ModernEngineering/raw-sql-superstore/blob/main/lab-5.md.

Before going into SQL, practice wireframing your JOINs on a piece of paper.

Solo Exercise:

JOINing Single Tables | Solution

Solution Query

```
SELECT
  o.order_id
  , c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
LIMIT 100;
```

JOINing Multiple Tables

You can also JOIN multiple tables together. Here is an example — notice that we have *two* JOIN statements.

Syntax: JOIN syntax restarts when you add on a new table:

```
SELECT a.field3, a.field4, b.field1, c.field4
FROM table1 a
    JOIN table2 b ON a.field1 = b.field1
    JOIN table3 c ON a.field2 = c.field1
ORDER BY b.field1
```


Wireframing JOINs | Multiple Tables

Secondary Table 1

returns r	
order_id	**reason_returned**
AE-2016-1308551	Not Given
AE-2016-1522857	Not Needed

ON

Primary Table

orders o	
order_id	**customer_id**
AE-2016-1308551	JR-16210
AE-2016-1522857	KM-16375
....	....

Secondary Table 2

customers c	
customer_id	**customer_name**
JR-16210	Justin Ritter
KM-16375	Katherine Murray

Partner Exercise:

JOINing Multiple Tables

10 minutes

Using **Orders** as our primary table, JOIN *both* the **Returns** *and* the **Customers** tables.

Before going into SQL, practice wireframing your JOINs on a piece of paper.

Your query should:

1. Include **order_id** from the Orders table, **customer_name** from the Customers table, and **reason_returned** from the Returns table.
2. Limit results to 100 rows.

Partner Exercise:

JOINing Multiple Tables | Solution

Solution Query

```
SELECT
   o.order_id
   ,c.customer_name
   ,r.return_date
FROM orders o
     JOIN customers c ON o.customer_id = c.customer_id
     JOIN returns r ON o.order_id = r.order_id
LIMIT 100;
```

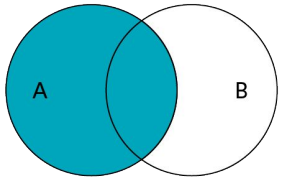
Partner Exercise:

JOINing Multiple Tables | Data Output

Desired Data Output

*	order_id	customer_name	return_date
1	AE-2019-1711936	Greg Hansen	2019-12-14
2	AE-2019-2092798	Greg Hansen	2019-11-29
3	AE-2019-2170363	Greg Hansen	2019-12-29
4	AE-2019-2262642	Greg Hansen	2020-01-04
5	AE-2019-2343602	Greg Hansen	2020-01-05
6	AE-2019-288592	Greg Hansen	2019-12-28
7	AE-2019-2952905	Greg Hansen	2019-12-18
8	AE-2019-3001630	Greg Hansen	2020-01-17
9	AE-2019-3369522	Greg Hansen	2019-11-29
10	AE-2019-3800683	Greg Hansen	2019-12-29
11	AE-2019-3959747	Greg Hansen	2019-12-17
12	AE-2019-4016062	Greg Hansen	2019-12-18
13	AE-2019-4579873	Greg Hansen	2020-01-09
14	AE-2019-4844787	Greg Hansen	2019-11-30
15	AE-2019-5196817	Greg Hansen	2019-12-31

LEFT JOINs



LEFT JOIN loads all entries that appear in the first table with NULLs where there is no match.

people

id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	5

vehicles

id	vehicle_name
1	Explorer
2	Civic
3	Corolla
4	Impala

id	name	vehicle_id	vehicle_name
1	Janet	3	Corolla
2	Emily	4	Impala
3	Yoko	5	NULL

Guided Walk-Through:
Let's Create a LEFT JOIN!

Let's revisit the query we wrote earlier that JOINs the Orders and Returns tables. We want to find all orders and **return information if it exists**. How should we JOIN these two tables?

Orders		
order_id	**order_date**	**ship_date**
AE-2016-1308551	2016-09-28	2016-10-02
AE-2016-1522857	2016-09-04	2016-09-09

+

Returns	
order_id	**return_date**
AE-2019-1711936	2019-12-14
AE-2019-2092798	2019-11-29

Guided Walk-Through:
Creating a LEFT JOIN

Knowing that we want to keep all entries that appear in the Orders table, we'll add a LEFT JOIN that designates Orders as the first table. Here is our query:

```
SELECT
  o.order_id
  ,r.return_date
FROM orders o
    LEFT JOIN returns r ON o.order_id = r.order_id
LIMIT 100;
```


Partner Exercise:

How Do We JOIN This? | Challenge

10 minutes

Superstore is developing a training program to help salespeople reduce the likelihood of returns. To do so, Superstore wants to interview salespeople (each salesperson has a region) who have processed higher volumes of returns in the past. You're generating a list of salespeople and return reasons (including NULL returns!). With your partner, discuss what type of JOIN(s) you will use. Be ready to explain why.

Orders			Returns		Region	
order_id	**order_date**	**ship_date**	**order_id**	**return_date**	**country**	**region**
AE-2016-1308551	2016-09-28	2016-10-02	AE-2019-1711936	2019-12-14	Benin	EMEA
AE-2016-1522857	2016-09-04	2016-09-09	AE-2019-2092798	2019-11-29	Morocco	EMEA

Partner Exercise:

How Do We JOIN This? | Solution

This aggregate is run *after* the JOIN on the Returns and Regions tables is complete.

```
SELECT
  rg.salesperson
  ,r.reason_returned
  ,COUNT(o.order_id) AS count_of_returns
FROM orders o
JOIN regions rg ON o.region_id = rg.region_id
LEFT JOIN returns r ON o.order_id = r.order_id
GROUP BY 1, 2
ORDER BY 3 DESC
LIMIT 100;
```

Solo Exercise:

Over to You | JOINS

20 Minutes

Follow the instructions in https://git.generalassemb.ly/ModernEngineering/raw-sql-superstore/blob/main/lab-6.md.

Solo Exercise:

Over to You | Solution

1) **How many orders were from the consumer segment?**

```
SELECT COUNT(*) FROM orders o INNER JOIN customers c ON o.customer_id =
c.customer_id WHERE segment ILIKE '%consumer%';
```

1) **What is the total value of consumer orders?**

```
SELECT SUM(o.sales) FROM orders o JOIN customers c
ON o.customer_id = c.customer_id WHERE c.segment ILIKE '%consumer%';
```

1) **Which segment has the highest average sale?**

```
SELECT c.segment, AVG(o.sales) FROM orders o JOIN customers c ON
o.customer_id = c.customer_id GROUP BY 1 ORDER BY 2 DESC;
```

Solo Exercise:

Over to You | Solution (Cont.)

4) **Which segment is responsible for the highest number of returns?**

```
SELECT c.segment, COUNT(*) FROM orders o JOIN customers c ON o.customer_id
= c.customer_id  JOIN returns AS r ON r.order_id = o.order_id GROUP BY 1
ORDER BY 2 DESC;
```

5) **Who is our top-selling salesperson in sales?**

```
SELECT r.salesperson, SUM(o.sales) FROM orders o JOIN regions r ON
o.region_id = r.region_id GROUP BY 1 ORDER BY 2 DESC;
```

6) **Did we have any unsold products?**

```
SELECT * FROM products p LEFT JOIN orders o ON p.product_id = o.product_id
WHERE o.product_id IS NULL;
```

JOINing Tables

UNIONs

Discussion:
Why UNIONs?

As we learned earlier, UNIONs combine rows from multiple tables with the same columns. In what scenarios will we use a UNION instead of a JOIN?

carpoolers		
id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	4

monthly_parkers		
id	name	vehicle_id
2	Emily	4
4	Ali	5
5	Ray	1

id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	4
4	Ali	5
5	Ray	1

UNION Syntax

Let's look at some simple mock syntax for a **UNION**:

```
SELECT field1
        FROM table1
UNION
SELECT field1
        FROM table2
```


Guided Walk-Through:
Exploring Examples of UNIONs

A UNION takes a single column or collection of columns and “stacks” them on top of each other. A common use case is if we have similar data between two tables and want to UNION those two tables together.

For illustration purposes, we’ll be using the following sample HR tables:

current_employees			
id	**first_name**	**last_name**	**salary**
2	Gabe	Moore	50000
3	Doreen	Mandeville	60000
5	Simone	MacDonald	55000

retired_employees			
id	**first_name**	**last_name**	**salary**
7	Madisen	Flateman	75000
11	Ian	Paasche	120000
13	Mimi	St. Felix	70000

Guided Walk-Through:

Creating a UNION for Two Tables

When you want to combine the two tables and both have the same columns, you can use a UNION with a SELECT *:

```
SELECT *
FROM current_employees
UNION
SELECT *
FROM retired_employees
```

id	first_name	last_name	salary
2	Gabe	Moore	50000
3	Doreen	Mandeville	60000
5	Simone	MacDonald	55000
7	Madisen	Flateman	75000
11	Ian	Paasche	120000
13	Mimi	St. Felix	70000

Guided Walk-Through:

Creating a UNION for Two Tables (Cont.)

You can also UNION tables on only columns. These columns must match data types but don't have to represent the same data. What happened in the table below? And where do the resulting headers come from?

```
SELECT first_name,
last_name
FROM current_employees
UNION
SELECT last_name,
first_name
FROM retired_employees
```

first_name	last_name
Gabe	Moore
Doreen	Mandeville
Simone	MacDonald
Flateman	Madisen
Paasche	Ian
St. Felix	Mimi

Rules for Using UNIONs

Remember these rules when using UNIONs:

- You *must* match the number of columns, and they *must* be of compatible data types.
- You can only have one **ORDER BY** at the bottom of your full SELECT statement.
- UNION removes *composite* duplicates.
- UNION ALL allows duplicates.

Partner Exercise:

Carmen Sandiego SQL Lab

120 minutes

Fork and Clone the Carmen Sandiego Lab here:

https://ga.co/4572mlh

